UNIVERSITAT POLITECNICA DE
CATALUNYA

PARALLELISM

Lab 5: Geometric (data) decomposition: heat diffusion equation

Roger Vilaseca Darne and Xavier Martin Ballesteros
PAR4110

7th June 2019, Q2

Contents

1 Introduction

In this last 3 laboratory sessions we have had to apply all the concepts we have learned during the course to parallelise the different functions that simulate heat diffusion in a solid body.

The two pictures below show the resulting heat distribution when two heat sources are placed in the borders of the 2D solid (one in the upper left corner and the other in the middle of the lower border). The first image has been computed using the *Jacobi* solver, which uses two matrices to do the computations (one is auxiliar). The second, on the other hand, is calculated using the *Gauss-Seidel* solver, which only uses one matrix.

Figure 1: Heat distribution with the *Jacobi* solver.

Figure 2: Heat distribution with the *Gauss-Seidel* solver.

In the first session we have used *Tareador* to analyse the dependences that exist in both solvers, and how we could guarantee these dependences when using *OpenMP*.

In the next two sessions, we have parallelised both solvers. Firstly, we have begun with the *Jacobi* solver, which has been easier to parallelise because it has less dependencies. Secondly, we have parallelised the *Gauss-Seidel* solver using new *OpenMP* clauses that we had not used before.

Finally, we have compared the two performances of the solvers using the execution time and speedup plots.

2 Analysis with *Tareador*

In this section we have used the *Tareador* tool to analyse the possible parallelism strategies that we can use in order to parallelise both Jacobi and Gauss-Seidel solvers.

We have mainly focused on the data dependences that appear and how will we protect them in our parallel *OpenMP* code. To explore the dependences, we have used a much finer task decomposition: one task for each iteration of the body of the most innerloop.

2.1 *Jacobi* Solver

This solver uses an auxiliar matrix **utmp** to write the resulting values of the computation in the most innerloop of the body. For each element in the matrix **u**, it takes the values of the element on its top and bottom and on its left and right, and does some arithmetic operations.

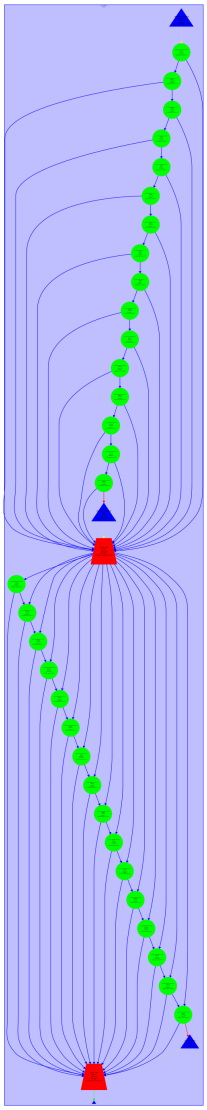
The modified code using the *Tareador* tool is shown below. However, the code can be found in *jacobi-tareador.c* file inside the codes directory.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex,
                     unsigned sizey)
{
    double diff, sum=0.0;

    int howmany=1;
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
           tareador_start_task("jacobi_innermost_task");
           utmp[i*sizey+j]= 0.25 * ( u[ i*sizey     + (j-1) ]+  // left
                                     u[ i*sizey     + (j+1) ]+  // right
                                     u[ (i-1)*sizey + j     ]+  // top
                                     u[ (i+1)*sizey + j     ]); // bottom
           diff = utmp[i*sizey+j] - u[i*sizey + j];
           sum += diff * diff;
           tareador_end_task("jacobi_innermost_task");
        }
      }
    }
    return sum;
}
```

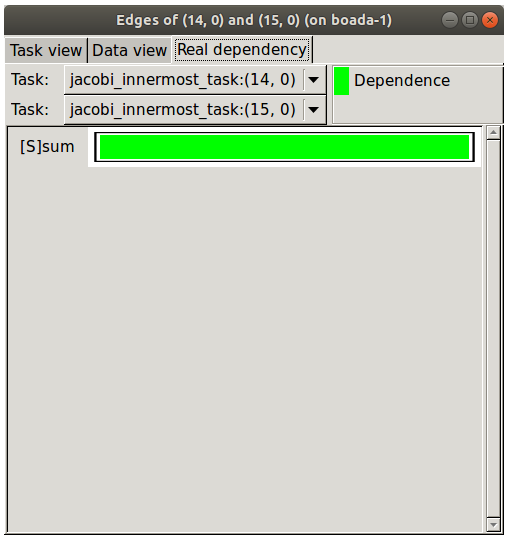
Figure 3: Code for the task decomposition for relax_jacobi function.

With this modified version of the code we can obtain the task decomposition graph (TDG), which can be found in Figure 4a. Moreover we can see that there exist some kind of data dependencies between tasks. To know which variable is the responsible of these dependences, we have right clicked in an edge between two

jacobi_innermost_task nodes (green) >> Dataview >> Edge >> Real dependency. The obtained results are shown in Figure 4b.



(a)



(b)

Figure 4: (a) Task decomposition graph for the Jacobi solver, (b) Real data dependences in the Jacobi solver.

Now, we know that the **sum** variable creates the dependences for the Jacobi solver. Novertheless, we have made use of some *Tareador* calls that temporarily filter the analysis for the variable sum, causing the serialization and obtaining a new task graph (Figure 6). The modified fragment of the code can be found in *jacobi-tareador-disable-sum.c* file in the codes directory.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex,
                     unsigned sizey)
{
    ...
    for (int blockid = 0; blockid < howmany; ++blockid) {
      ...
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
            tareador_start_task("jacobi_innermost_task");
            ...
            tareador_disable_object(&sum);
            sum += diff * diff;
            tareador_enable_object(&sum);
            tareador_end_task("jacobi_innermost_task");
        }
      }
    }

    ...
}
```

Figure 5: Code for the task decomposition for relax_jacobi function temporarily filtering the analysis of the sum variable.

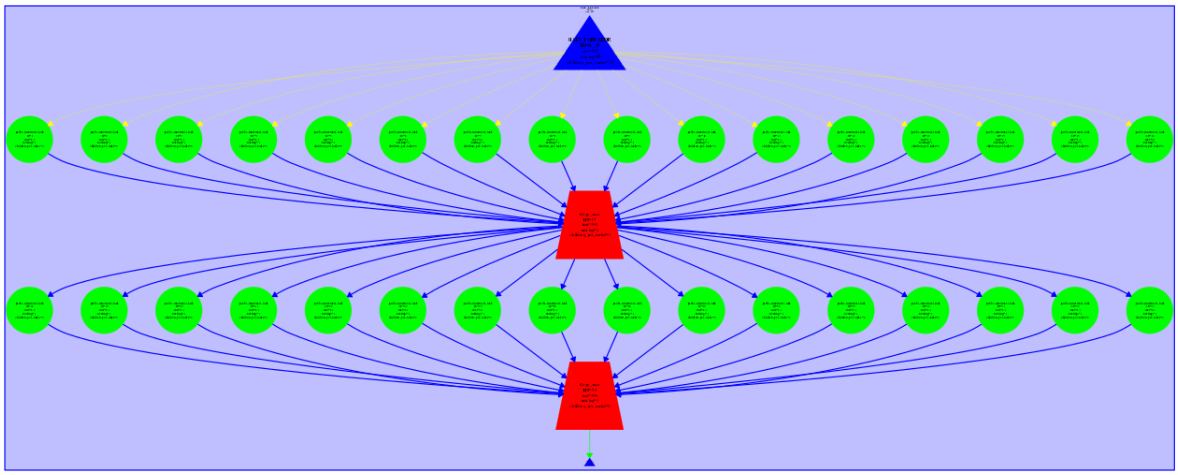


Figure 6: Task decomposition graph of the Jacobi solver temporarily filtering the analysis of the sum variable.

Now, we can see that there is any dependency between tasks of the most inner-loop. Hence, we are increasing the parallelism. We think that the *reduction(+:sum)* clause would be a good option to parallelise the code using *OpenMP* directives.

2.2 *Gauss-Seidel* Solver

The Gauss-Seidel Solver does no longer use an auxiliar matrix. It writes in the position of the matrix where it reads the values. As in the previous solver, it takes

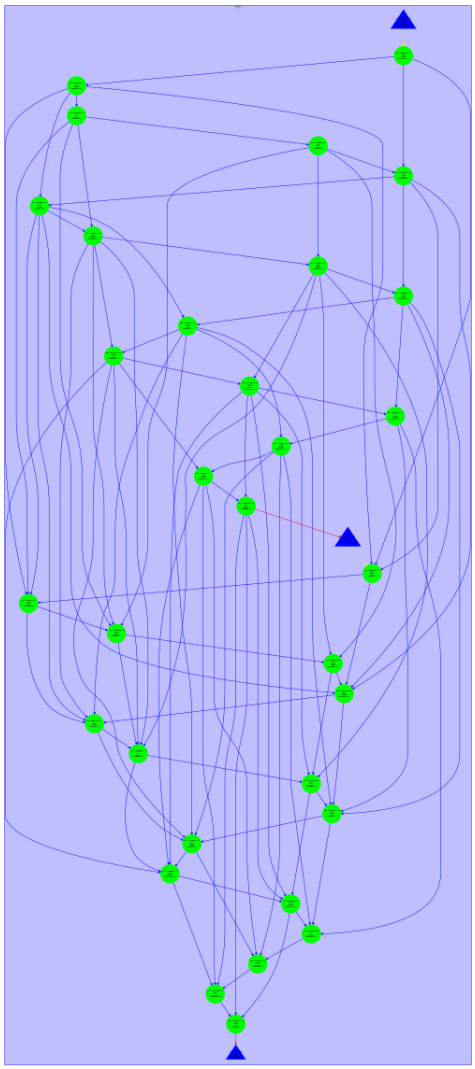
the values of the element on its top and bottom and on its left and right, and does some arithmetic operations.

The modified code can be found in *gauss-seidel-tareador.c* file in the codes directory.

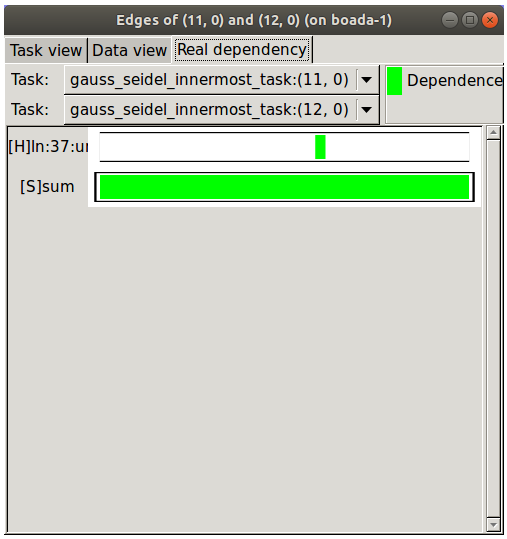
```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;

    int howmany=1;
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
            tareador_start_task("gauss_seidel_innermost_task");
            unew= 0.25 * ( u[ i*sizey   + (j-1) ]+  // left
                           u[ i*sizey   + (j+1) ]+  // right
                           u[ (i-1)*sizey  + j     ]+  // top
                           u[ (i+1)*sizey  + j     ]); // bottom
            diff = unew - u[i*sizey+ j];

            sum += diff * diff;

            u[i*sizey+j]=unew;
            tareador_end_task("gauss_seidel_innermost_task");
        }
      }
    }

    return sum;
}
```

Figure 7: Code for the task decomposition for relax_gauss function.

The task decomposition graph is shown in Figure 8a. We can observe that it has also some dependencies. Using the same procedure than in the previous section, we got the Real dependencies for this new solver (Figure 8b).



(a)



(b)

Figure 8: (a) Task decomposition graph for the Gauss-Seidel solver, (b) Real data dependences in the Gauss-Seidel solver.

We can see that this solver has two real dependencies: variable **sum** and some **positions of the matrix**. In this section, we will only show the TDG when disabling only the sum variable. However, the TDG of both variables disables can be found in the Annex section 6.1.

This new version of the code can be found in *gauss-seidel-disable-sum.c*, in the codes directory.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    ...
    for (int blockid = 0; blockid < howmany; ++blockid) {
      ...
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
            tareador_start_task("gauss_seidel_innermost_task");
            ...
            tareador_disable_object(&sum);
            sum += diff * diff;
            tareador_enable_object(&sum);
            ...
            tareador_end_task("gauss_seidel_innermost_task");
        }
      }
    }
    ...
}
```

Figure 9: Code for the task decomposition for relax_gauss function temporarily filtering only the analysis of the sum variable.

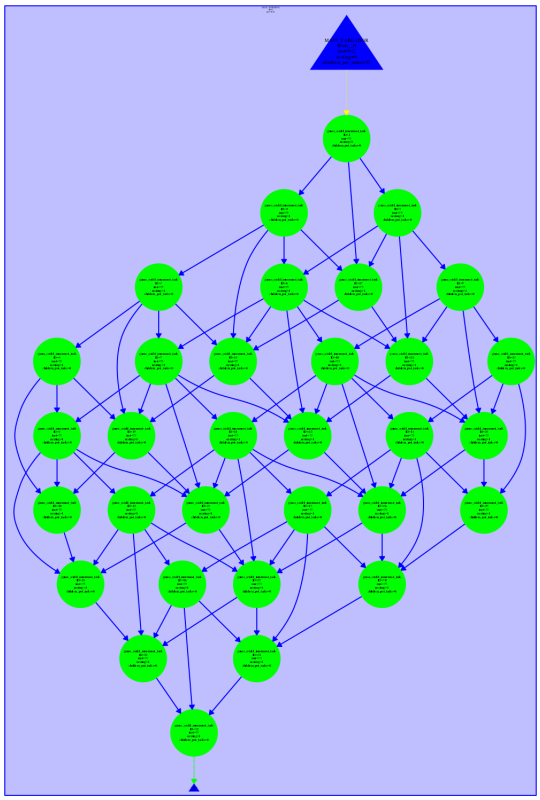


Figure 10: Task decomposition graph of the Gauss-Seidel solver temporarily filtering only the analysis of the sum variable.

Disabling temporarily the variable sum we see that we increase the parallelism.

Again, we think that the *reduction(+:sum)* clause would be good to porallelise the code using *OpenMP* directives. Moreover, we can use the *ordered* clause in order to avoid data races and respect the real dependencies we have seen (apply the doacross technique).

3 Parallelization of *Jacobi* with OpenMP `parallel`

In this section, we had to understand how Jacobi solver worked and afterwards, parallelize it in order to increase the speedup of the program.

3.1 Understanding the code

We had given some definitions already created in order to facilitate us the readability of the code. This functions are:

```
#define lowerb(id, p, n) ( id * (n/p) + (id < (n%p) ? id : n%p) )
#define numElem(id, p, n) ( (n/p) + (id < (n%p)) )
#define upperb(id, p, n) ( lowerb(id, p, n) + numElem(id, p, n) - 1 )
#define min(a, b) ( (a < b) ? a : b )
#define max(a, b) ( (a > b) ? a : b )
```

Figure 11: Given functions

The functions **lowerb** and **upperb** return the first and the last index of a vector partiton, when it is given an id (the partition number), p (the number of partitions) and n (the size of the vector).

The function **numElem** returns the number of elements that are in a partition of a vector. Having the same entries as before.

Finally, **min** and **max** functions return the minimum and maximum number between two values respectively.

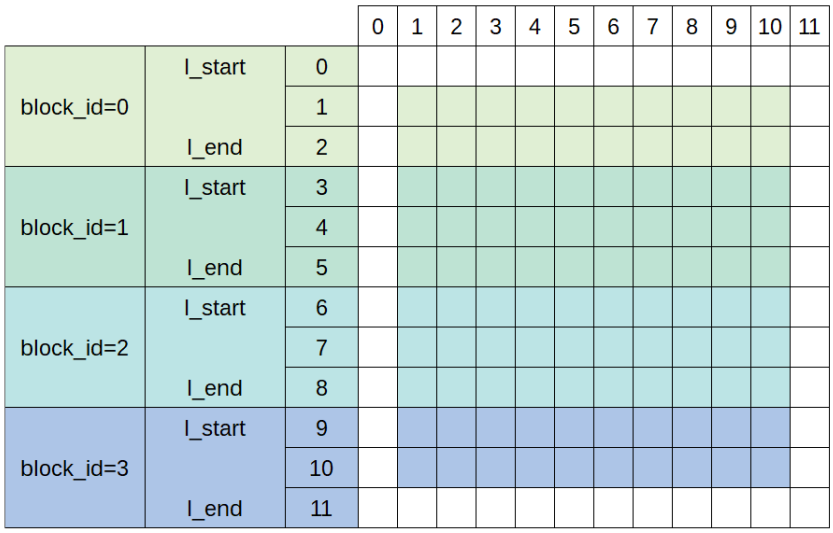


Figure 12: Geometric data decomposition.

In the geometric data decomposition we can see how the block_id is distributed in a matrix (12 × 12). Assuming that the code is using a vector to represent the matrix, the size of the vector would be 144.

The first and last row and column (white cells in the matrix) are not accessed due to the fact that to do the computation for each element of the matrix, it needs to have an element above, below, right and left with respect to him, as we will see in the following section.

3.2 Parallelization of *Jacobi*

In this part we had to parallelize the code considering that we wanted to create 4 blocks, without using the *#pragma omp parallel for* clause in the code. To do that we have modified the relax_jacobi function, which can be found in *solver-omp-jacobi-S2-2.c* file.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;

    int howmany = 4;
    #pragma omp parallel reduction(+: sum) private(diff)
    {
        int blockid = omp_get_thread_num();
        int i_start = lowerb(blockid, howmany, sizex);
        int i_end = upperb(blockid, howmany, sizex);
        for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
            for (int j=1; j<= sizey-2; j++) {
                utmp[i*sizey+j]= 0.25 * ( u[ i*sizey + (j-1) ]+ // left
                                          u[ i*sizey + (j+1) ]+ // right
                                          u[ (i-1)*sizey + j ]+ // top
                                          u[ (i+1)*sizey + j ]);// bottom
                diff = utmp[i*sizey+j] - u[i*sizey + j];
                sum += diff * diff;
            }
        }
    }

    return sum;
}
```

Figure 13: Code for parallellization of relax_jacobi function

As we can see we have divided the matrix in 4 parts. This will cause that, if we execute the code with 8 threads only the first 4 threads will have a valid blockid number, so the other 4 threads will recieve an out of bound *i_start* index. Thus, the remaining threads will only execute the first 3 lines of the parallel part and will not enter inside the two fors.

On the other hand, we have used the *reduction(+: sum)* clause to respect the dependences we have seen when using *Tareador*.

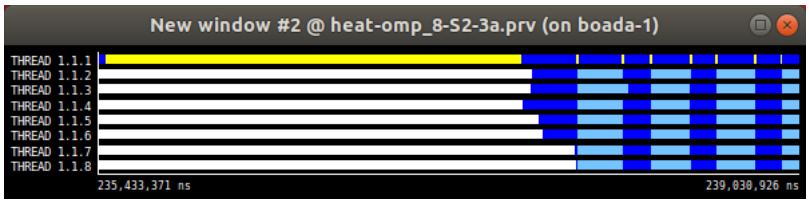


Figure 14: Traces obtained when executing the relax_jacobi function with only 4 threads doing the computations.

In order to solve this issue, we have modified a bit the *relax_jacobi* function. Now, the variable **howmany** has the total number of threads we want to use. Hence, the code will divide the matrix taking into account all the operative threads, and not only the first four as before. This new version of the code can be found in file *solver-omp-jacobi-S2-3.c*, in the codes directory.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;

    #pragma omp parallel reduction(+: sum) private(diff)
    {
        int howmany = omp_get_num_threads();
        ....
    }

    return sum;
}
```

Figure 15: Improved code of relax_jacobi function.

With this little change, we can see that we obtain a better parallelization of the program. In other words, we take more advantage of the potential parallelism of the code. If we compare the execution time of the trace obtained with this new version of the code (16) with the previous trace (14), we see that it spends less time in the parallel part. Hence, it goes faster, because the other 4 threads now do some computation in the matrix.

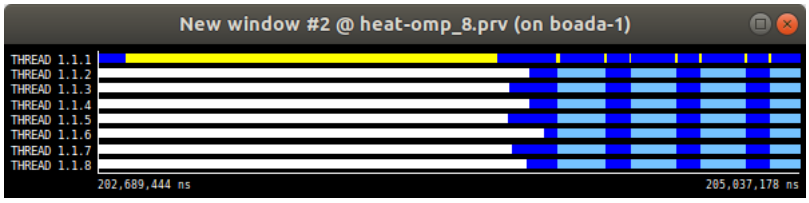


Figure 16: Improved result of the execution of the relax_jacobi function with the change in the variable howmany.

However, this was not enough parallelism. We realised that the function *copy_mat* had also a big impact in the code, so we decide to parallellize it. The code can be found in *solver-omp-jacobi-S2-4.c* file.

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
    #pragma omp parallel for collapse(2)
    for (int i=1; i<=sizex-2; i++)
        for (int j=1; j<=sizey-2; j++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
}
```

Figure 17: Code for the parallellization of the copy_mat function.

Now, we have obtained the best results so far regarding to parallelization. Each thread copies a fragment of the input matrix. We can see in the following image that the light blue (Idle state) part has decreased, which means that threads are now doing more work.

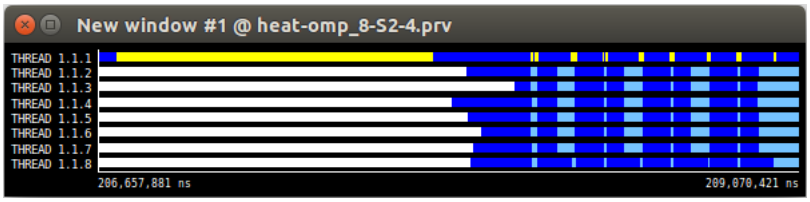


Figure 18: Improved Jacobi Schedule with copy_mat parallellization.

Finally, in order to test the scalability of our program, we have executed the script *submit-strong-omp.sh* to obtain the following speedup and scalability plots.

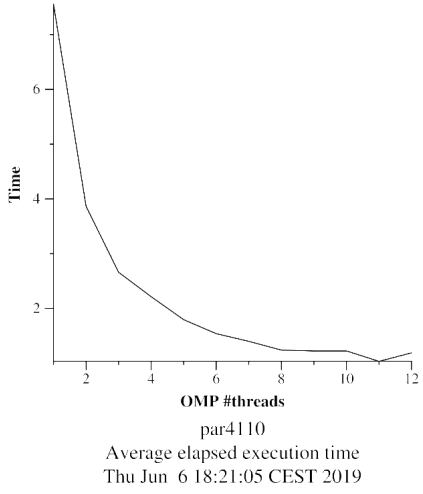


Figure 19: Execution time plot when varying the number of processors for the Jacobi solver.

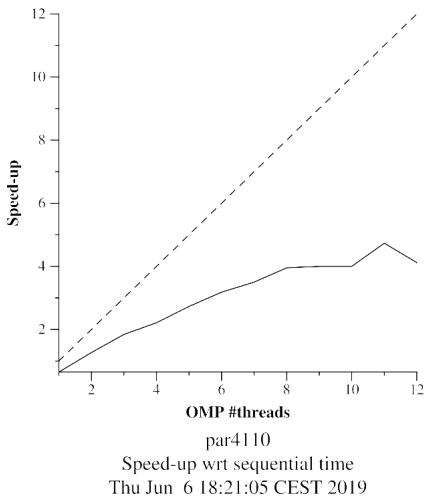


Figure 20: Speedup plot when varying the number of processors for the Jacobi solver.

As we can see the performance increases together with the number of threads at first. But when we have a large number of threads the results shows a bit stagnation of this improvement, probably due to the creation and termination of threads and the synchronizations between them.

4 Parallelization of *Gauss-Seidel* with OpenMP `ordered`

In this section we have parallelized the Gauss-Seidel solver using #pragma omp for and its **ordered** clause.

The important part with this solver was to decide how we would synchronize the parallel execution of the rows assigned to each processor in order to guarantee the dependences that we detected with *Tareador*.

At first, we decided to divide the matrix as in figure 12. However, we realized that with that geometric data decomposition, the code would be executed exactly the same than in the sequential version because thread 1 would have had to wait until thread 0 terminated... Consequently, we had to think a new data decomposition.

The solution was to divide the matrix into blocks not only dividing the rows but the columns. Hence, when thread 0 finishes the first block, it continues excuting its new block whereas thread 1 can start executing its first block, and so on.

Then, we used again the **reduction(+:sum)** clause to improve the performance of the program without creating data races. Besides, we have also implemented the **doacross** technique using the **ordered(2)** clause, **#pragma omp ordered depend(sink: ...)** and **#pragma omp ordered depend(source)**. Each block can only be executed when blocks (i-1, j) and (i, j-1) terminate.

The code can be found in file *gauss-seidel-reduction-doacross.c* in the codes directory.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey) {
    double unew, diff, sum=0.0;

    #pragma omp parallel private(unew, diff) reduction(+: sum)
    {
    int howmany = omp_get_num_threads();

    #pragma omp for ordered(2)
    for (int blockid_i = 0; blockid_i < howmany; ++blockid_i) {
        for (int blockid_j = 0; blockid_j < howmany; ++blockid_j) {

            int i_start = lowerb(blockid_i, howmany, sizex);
            int i_end = upperb(blockid_i, howmany, sizex);
            int j_start = lowerb(blockid_j, howmany, sizey);
            int j_end = upperb(blockid_j, howmany, sizey);

            #pragma omp ordered depend(sink: blockid_i-1,blockid_j)
            for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
                for (int j=max(1, j_start); j<= min(sizey-2, j_end); j++) {
                    unew= 0.25 * ( u[ i*sizey   + (j-1) ]+  // left
                                   u[ i*sizey   + (j+1) ]+  // right
                                   u[ (i-1)*sizey   + j ]+  // top
                                   u[ (i+1)*sizey   + j ]); // bottom
                    diff = unew - u[i*sizey+ j];
                    sum += diff * diff;
                    u[i*sizey+j]=unew;
                }
            }
            #pragma omp ordered depend(source)
        }
    }
    }

    return sum;
}
```

Figure 21: Modified code of the function relax_gauss.

Afterwards, we have used the *Paraver* tool to see if the flow of the program matched our expectations. Due to the use of the **ordered** clause, we can see in Figure 22 that there is a lot of synchronization between threads. As we said before, thread 1 has to wait until thread 0 ends its first block, thread 2 has to wait until thread 1 terminated its first block... As a consequence, we see a kind of stairs at the beginning of each thread, in blue; and at the end of each thread execution, in red (the first thread will be the first to end, and so on).

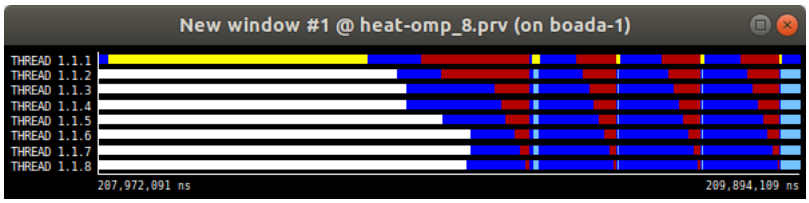


Figure 22: Traces obtained when executing the new version of relax_gauss function.

Finally, we have analysed the scalability of our parallelization. We can see in Figures 23 and 24 that the results are worse than the plots in the Jacobi solver. The main reason is that this solver has more dependencies than Jacobi, so it has more synchronizations. Thus, it cannot have the same or better results.

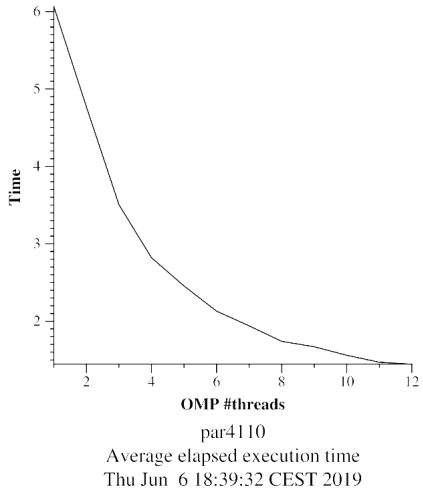


Figure 23: Execution time plot when varying the number of processors for the Gauss-Seidel solver.

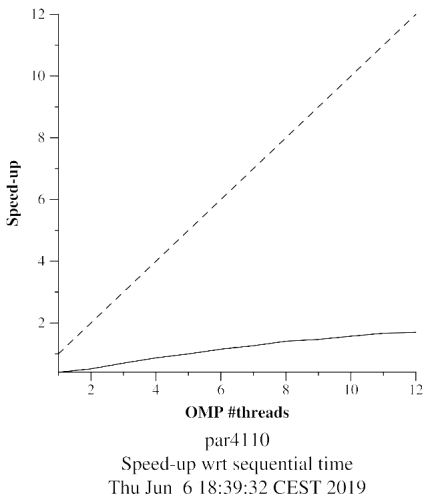


Figure 24: Speedup plot when varying the number of processors for the Gauss-Seidel solver.

Encara FALTA mirar a baix.

How can you control in your code the trade-off between computation and synchronization? Is there an optimum value for the ratio between computation and synchronization? For the execution with 8 threads, explore possible ratios and plot how the execution time varies.

Analyse the speed–up (strong scalability) plot that has been obtained for the different numbers of processors, reasoning about the performance that is observed and including captures of Paraver windows to justify your explanations.

Finally explain how did you obtain the optimum value for the ratio computation/synchronization in the parallelization of this solver for 8 threads.

5 Conclusions

In these 3 session of laboratory we have seen different algorithms to simulate the heat diffusion in a solid body.

On the one hand, we have seen the *Jacobi* solver. This uses an auxiliar matrix where it writes the results of the computations. Thus, we have seen that it only has dependency on the variable **sum**. This dependency was easy to guarantee using the *reduction(+:sum)* clause in the *OpenMP* library.

On the other hand, the *Gauss-Seidel* solver does not need an auxiliar matrix. It writes into the same matrix where it reads the elements used in the calculus. Consequently, it has more dependencies than the other solver. The new dependencies have been guaranteed in the parallel code using the **doacross** technique (*#pragma omp for ordered(2)*, *#pragma omp ordered depend(sink:...)* and *#pragma omp ordered depend(source)*).

We have seen that the first solver needs more memory but has less dependencies, while the second solver is the other way round. Hence, the programmer that will use these algorithms will have to decide depending on his needs which solver to use.

6 Annex

6.1 Annex I: Task dependency graph when also disabling temporarily some positions of the matrix

In this section we will see the TDG created when disabling all the variables that create some kind of dependency in the Gauss-Seidel solver. The code can be found in *gauss-seidel-disable-sum-and-matrix-positions.c*, inside the codes folder.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    ...
    for (int blockid = 0; blockid < howmany; ++blockid) {
      ...
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
            tareador_start_task("gauss_seidel_innermost_task");

            tareador_disable_object(&u[ i*sizey    + (j-1) ]); // left
            tareador_disable_object(&u[ (i-1)*sizey    + j     ]); //top
            unew= 0.25 * ( u[ i*sizey  + (j-1) ]+  // left
                           u[ i*sizey  + (j+1) ]+  // right
                           u[ (i-1)*sizey  + j     ]+  // top
                           u[ (i+1)*sizey  + j     ]); // bottom
            diff = unew - u[i*sizey+ j];
            tareador_enable_object(&u[ i*sizey + (j-1) ]);
            tareador_enable_object(&u[ (i-1)*sizey + j     ]);

            tareador_disable_object(&sum);
            sum += diff * diff;
            tareador_enable_object(&sum);

            ...
            tareador_end_task("gauss_seidel_innermost_task");
        }
      }
    }
    ...
}
```

Figure 25: Code for the task decomposition for relax_gauss function temporarily filtering the analysis of the sum variable and some positions of the matrix.

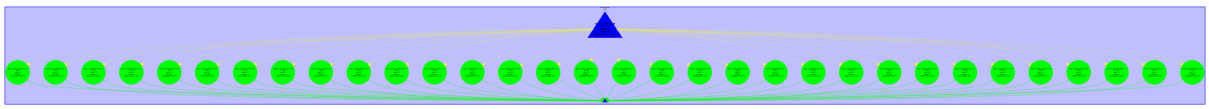


Figure 26: Task decomposition graph of the Gauss-Seidel solver temporarily filtering the analysis of the sum variable and some positions of the matrix.